A MULTI-LAYER DEPENDENCY MODEL FOR ANALYSIS OF SAFETY-CRITICAL EMBEDDED SYSTEMS

BY

MARYAM RAHMANIHERIS

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2011

Urbana, Illinois

Advisor:

Professor Lui Sha

Abstract

Safety-critical embedded-system designs are typically both complex and expensive. Domains, such as medical devices, however, require safety but also demand affordability. However, conventional safety and reliability engineering methods, including redundancy or conventional dependency analysis, often lead to expensive and complex system designs.

In this work, we propose a multi-layer dependency framework to analyze safety-critical systems. This framework captures fine-grained dependencies in safety-critical systems compared with traditional dependency graph analysis. Due to this new approach, we are able to verify the safety of systems with a reduced degree of redundancy, compared with conventional reliability engineering methods. To show the effectiveness of the multi-layer dependency framework, we apply it to four applications in the medical and control domains. These studies show a reduction in the complexity of the associated safety subsystems, which translates to both a reduction in cost and a reliability improvement for the safety subsystem. We specifically discuss the the applicability of our dependency framework to distributed medical systems where the conventional two-layer dependency model is unable to analyze the safety of complicated supervisory frameworks for such systems.

Acknowledgments

This project would not have been possible without the support of many people. Many thanks to my adviser, Lui Sha, who read my numerous revisions and offered research guidance and valuable discussions. Also thanks to my colleagues, Cheolgi Kim, and Stanely Bak, who offered help and support. And finally, thanks to my family who always offered support and love.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

As safety-critical embedded systems offer more functionalities and features, their complexity increases accordingly and ensuring system safety becomes extremely difficult. In safety engineering, it is common to partition the system into critical and non-critical subsystems and focus on improving the reliability and safety of the critical part and its safe interaction with the rest of the system. A great amount of work has been done in this area toward determining safety requirements in early stages of design and making systems safer [1], [2], [3], [4]. Reliability engineering methods are also used to predict and understand the reliability of safety subsystems [5], [6]. Another commonly used method is fault tree analysis to ensure no undesired event in the system can compromise the system-wide safety [7], [8], [9]. A stronger reasoning framework used to discover safety dependencies in the system is the dependency graph approach [10], [11], [12]. In this work we propose an improvement to conventional dependency graph analysis [12] where we introduce a multi-layer dependency model and its associated fault-propagation rules. The refined model can capture the characteristics of safety-critical embedded systems in a more precise manner. Dependency graphs play a major role in designing reliable and safe embedded systems. System safety dependency graphs can be used for rapid analysis of the design to ensure the safe composition of the system. The conventional dependency graph approach has been used to compose safety-critical embedded systems [13], [14], [15], [16]. In this work, we build upon the dependency graph approach by proposing a more general, multi-layer dependency model. The necessity of this model is illustrated in the following example system.

However, this conventional model has a coarse-grained classification of dependency relations. Consider a Medical Device Plug-and-Play (MDPnP) interoperability system for airway-laser surgery. We assume a version of this system with four components: a ventilator, an airway laser, a pulse oximeter, and a central supervisor. The devices are connected to the supervisor trough a communication network. The airway-laser operation is performed on the patient's face or neck area using a surgical laser when he is under anesthesia and breathing with the help of an endotracheal tube carrying air with a high concentration of oxygen. The vicinity of the tube to the laser beam bears the risk of accidental tube ignition and subsequent airway fire[1]. Therefore, oxygen flow must be blocked while the laser is in use. However, this leads us to the second potential hazard which is brain damage. If the

[1]Each year, more than 600 surgical fires are reported. The most severe burns are internal and caused by burning breathing tubes [17].

oxygen flow is blocked for too long, the patient's blood-oxygen saturation can drop below the safe threshold and brain damage can occur.

It is crucial to have automatic supervisory systems coordinate the devices appropriately to ensure the patient's safety. In the system, all the necessary safety interlocks are implemented in the supervisor. Conventional dependency model analysis only models *depends* and *uses* relations. If the conventional dependency graph model is used, the supervisor will be modeled with a *depends* relation and therefore system safety fully depends on the supervisor to send the correct and synchronized commands to the devices. Hence, the communication network should always work and the supervisor software and all the underlying layers should never fail. It is possible to ensure system safety by using redundancy or custom-designed highly-reliable computing and communication equipment. However, both of these solutions are excessively expensive. The main question is: *Is it possible to reduce the need for redundant and expensive equipment while maintaining system safety?*

In this work, we introduce a multi-layer dependency framework by splitting the *depends* relation into a *timeliness* dependency and a *correctness* dependency. In safety-critical systems, there are some components that are only expected to deliver non-faulty service while they are alive. In other words, these components are fail-silent components and system safety only depends on their correctness. However, the challenge is how to model and analyze a system such as our example surgery scenario, where coordination among medical devices is needed, but both the central supervisor and network may crash (fail-silent). The full dependency relation in the conventional model does not capture the fail-silent concept.

We start by elaborating on the multi-layer dependency framework in Section 2. In this section, we assume a simplified supervisory system for our airway-laser surgical scenario related to the NASS framework [18]. We show that system safety only depends on the correctness of the supervisor by only making a few changes to the design. As a result, failure of the supervisor or network disconnection does not affect system safety. The interconnection networks can be unreliable and the supervisor, therefore, can be deployed on a lower-cost platform which may fail, such as one with a COTS (Commercial Off-the-Shelf) operating system and microprocessor. Thus, applying the multi-layer dependency framework to the system removes the need for redundant and expensive equipment.

A demo of NASS framework can be viewed at [19]. This framework is a medical supervisory framework for safe coordination of medical devices which is designed according to ICE standard [20]. This standard is developed by MDPnP group [21] with the goal of improving the safety and efficacy of patient care. The standard specifies the basic elements of an automatic medical supervisory system for interconnected medical devices. The authors in [22], [23] and [24] also focus on the necessity of systems of interconnected plug-and-play medical devices to improve system safety.

Chapter 2

Basic Approach

In this section we take a closer look at dependency relations between components of a system to differentiate two types of dependencies: *Correctness* and *Timeliness*. Then we introduce our dependency model that uses our fine-grained dependency categorization to obtain a system composition with reduced-complexity critical subsystem while maintaining the system safety. The goal of our work is to present a systematic approach to improve the reliability of safety subsystem without compromising system safety. In systems in which we can recognize these different types of dependencies, the complexity of safety-critical subsystem can be reduced significantly.

2.1 Motivation Example

We will build our dependency model and explain its benefits using an airway-laser surgery example. In this subsection, we provide an intuitive explanation of how multi-layer dependency model can improve the system design in terms of reliability and cost. We spend the next two subsections illustrating the conceptual model and how to use it to determine if the system composition is safe. Our dependency model can also be used to determine if the system can further be improved in terms of reliability of safety subsystem.

The airway-laser surgery system is composed of four components: ventilator, airway laser, pulse oximeter, and the supervisor. As seen in Fig.2.1 the devices are connected to the supervisor through a communication network. To prevent surgical fire, laser and ventilator should be synchronized by the supervisor. This process is performed through a series of handshakes between the supervisor and the devices. Fig.2.2.a and Fig.2.2.b illustrate this process for the conventional design. As we can see from Fig.2.2.b, when there is a network or supervisor failure, patient safety can be violated.

In contrast, in the design based on multi-layer dependency model we can have a safe and more reliable system composition just by attaching a simple timer to each medical device. As mentioned previously, we use a simplified version of timer-based approach to explain the need for a multi-layer dependency model. In the initial configuration of the system the ventilator is enabled and the oxygen flow is open but the laser is disabled. The following explains the steps of airway-laser surgery in a timer-based supervisory system:

1. To start the surgery, the supervisors reads the patient $SpO2$ level monitored by the pulse oximeter.
2. If the value is above a safe threshold then it sends a disable command to ventilator with a timer value.
3. The supervisor waits for an ACK form the ventilator.
4. After receiving the ACK form the ventilator, the supervisor waits for a safe amount of time to ensure that all the oxygen in the patient's airway has depleted before the laser becomes enabled to prevent fire.
5. The supervisor sends an enable command to airway laser along with its timer value.
6. The supervisor monitors patient's $SpO2$ level constantly.
7. If patient $SpO2$ drops below the safe threshold, the supervisor sends a disable command to laser.
8. The supervisor sends an enable command to ventilator to open the oxygen flow.
9. To resume the surgery, a safe amount of time should elapse and $SpO2$ reading should return to the safe range.
10. If the the supervisory loops becomes open in the middle of laser operation, first the laser deactivation timer and then ventilator activation timer expires.

As we can see in Fig.2.2.c, a failure of the supervisor or network disconnection incurs neither a fire nor a brain damage. The deactivation timer of the laser is set to expire earlier than the oxygen controller's enable timer. If the supervisor fails in the middle of the laser operation, both timers will expire, and the devices will come back to the default operations. Because the laser is turned off before the ventilator's activation, no fire hazard will occur even though the supervisor has failed in the middle. Moreover, a brain damage can also be prevented because the timer make the ventilator to be turned on before low oxygen saturation happens. The device timers should be implemented on a dedicated hardware to avoid the complex layers of software, operating system and microprocessor.

We also use Fault Tree analysis to demonstrate the behavior of the system which lead to the hazardous and unsafe situations. Safety is in fact a matter of system behavior that cannot be modeled by fault trees which are simply representations of casual chains. Therefore, we use Extended Fault Tree which (EFT) is the combination of classic fault Trees and State/Event semantics. The main goal is adding the capability of expressing state dependencies or temporal order of events. This is necessary to describe a system's behavior and analyze its reliability and safety-related issues. In EFT states describe conditions that last over a period of time whereas events are sudden phenomena, including state transitions. In EFT round rectangles represent states and events are denoted by solid

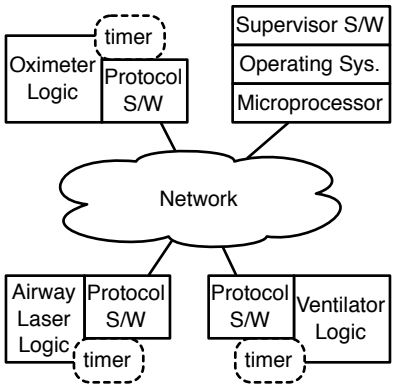


Figure 2.1. Overview of Airway-laser Surgery Supervision System

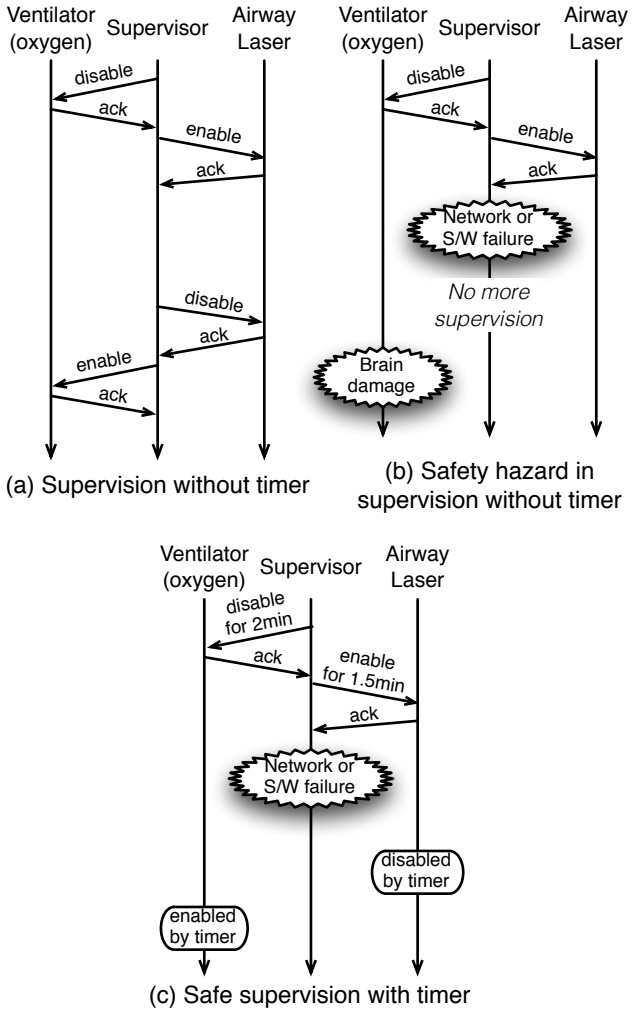


Figure 2.2. Handshaking between Supervisor and Medical Devices in Airway-laser Surgery

bars. Directed edges with bold arrowheads mark the casual edges and those with light arrowheads mark temporal edges. Fig. 2.3 shows the fault tree analysis for surgical fire and Fig. 2.4 shows the analysis of brain damage incident for the first design.

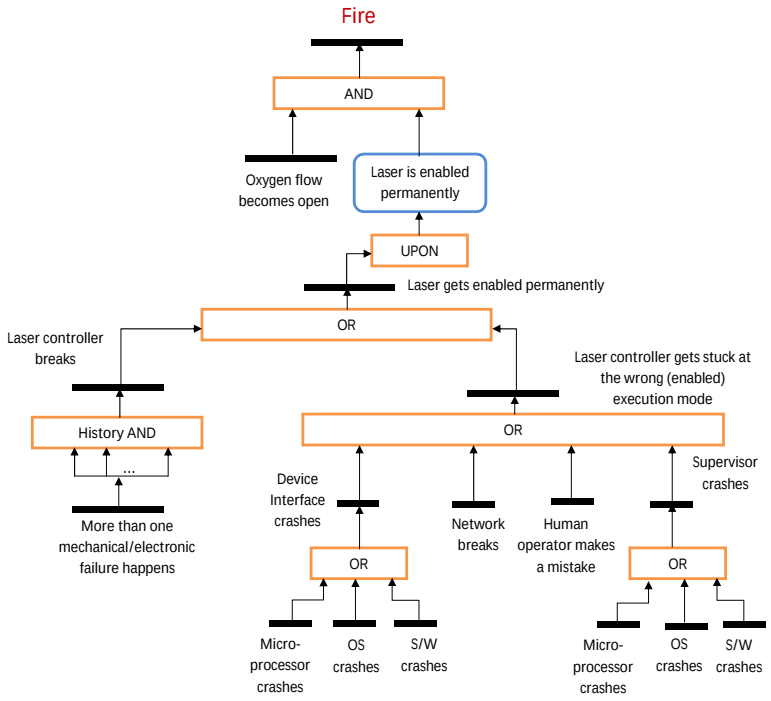


Figure 2.3. Fault Tree Analysis of Surgical Fire

In the first design, the critical part of the system includes the communication network and the supervisor software and all the underlying layers. In contrast, in our design based on the multi-layer dependency model, the system safety have only correctness dependency on the supervisor and have full dependency on device timers. Therefore, the critical part of the system only includes the simple timers implemented on dedicated hardware. Therefore, the new dependency model leads to a significant increase in reliability of safety-critical subsystems while all safety requirements are still satisfied. In fact, the new dependency framework leads to a much safer design. This is due to the fact that, in the new system composition the safety fully depends on simple timers implemented in hardware which has tremendously lower failure rate compared to complex and unreliable components such as communication network, operating system and microprocessor. In the first design the failure of any of complex components can endanger the system safety while in the second design the safety only depends on the simple timer. Fig. 2.5 shows the reduction in the fault tree of both of the adverse events in the second design. A large branch in both fault tree is substituted by a small event which is the failure of safety-critical hardware. The probability of failure of this

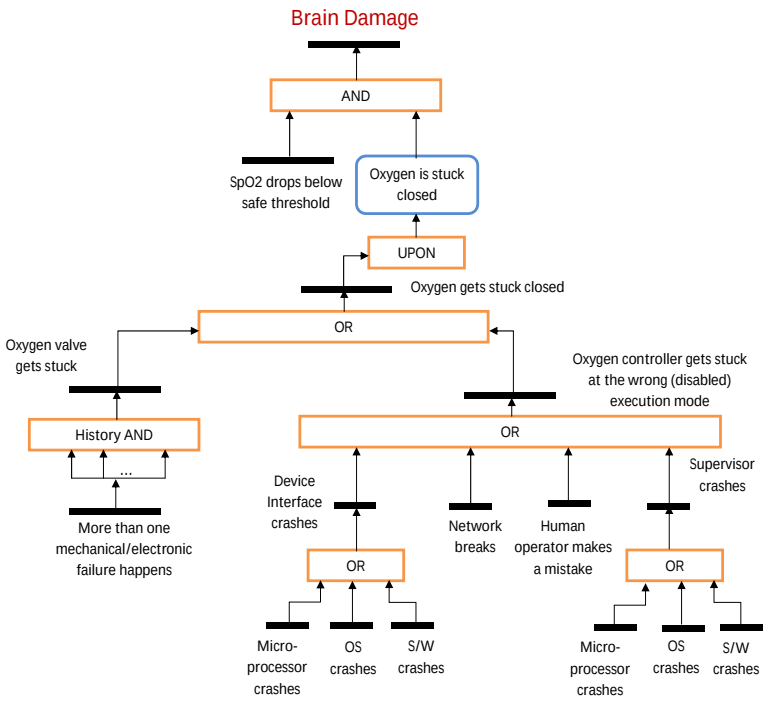


Figure 2.4. Fault Tree Analysis of Brain Damage in Patient

hardware component is multiple orders of magnitude smaller than the probability of failure of any of the complex and unverifiable components in the original branch such operating system or the communication network.

In the rest of this section, we use the two different designs of airway-laser surgery system to explain the limitations of conventional dependency model and introduce the multi-layer dependency model. We show how to determine if the system composition is safe and if the system can be optimized in terms of cost and reliability of safety subsystem using our refined dependency model.

2.2 Definitions

Ding et al. [12] introduce a dichotomy dependency framework in which the dependency relations between the components of a system is categorized into *use* and *depend*. Suppose component B delivers service to component A. Component A *use* component B if component A can function correctly and meets its safety requirements in spite of all possible faults in component B. This relation is represented by $A \xrightarrow{use} B$ in the dependency graph. Component A *depend* on component B if component A fails for any faulty service delivered by component B. This relation is represented by $A \xrightarrow{depend} B$ in the dependency graph.

According to this dependency model, The airway-laser surgery system depends on the communication network

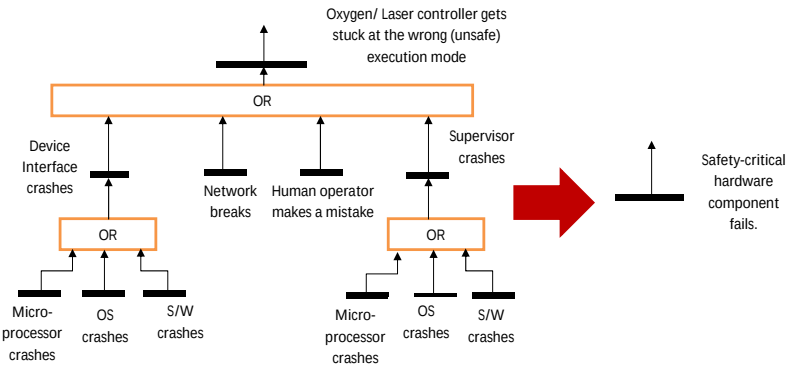


Figure 2.5. Fault Tree Reduction in the Design based on Multi-layer Dependency Model

and the supervisor software and all the underlying layers. If the system is denoted by S, then from the system safety perspective we have the followings:

- $S \xrightarrow{depend} SupervisorSoftware$
- $S \xrightarrow{depend} OperatingSystem$
- $S \xrightarrow{depend} Microprocessor$
- $S \xrightarrow{depend} CommunicationNetwork$

Now, we introduce a fine-grained categorization of dependency relations. We differentiate *correctness* and *full* dependencies encapsulated within *depend* relation. The study of safety-critical embedded systems shows that *use* and *depend* relations may not be a precise way of describing the dependency relations between the components of a system.

Definition1: Correctness Dependency Component A depends only on B's *Correctness*, if A can meet its safety specifications, even though data and/or commands from B are missing, early or late. However, component B's data or command are not incorrect if they arrive. This relation is represented by $A \xrightarrow{correct} B$ in the dependency graph. If component $A \xrightarrow{correct} B$, to formally verify the Correctness of A, the Correctness of component B must be verified as well.

Definition2: Timeliness Dependency component A is said to be dependent on *Timeliness* of component B, if component A depends on component B to be alive and functioning in a timely manner throughout the system lifetime. *Timeliness* dependency is a stronger relation than liveness dependency. In real-time domain the failure

to produce the results is as harmful as failure to produce the results on time. Therefore, we only consider the *Timeliness* dependency relation since it contains the liveness dependency inside itself.

Definition3: Full Dependency component A is said to be *fully* dependent on component B, if any fault in B will lead to As failure. In this case component A depends on component B to be live and functioning correctly and timely throughout system lifetime. In other words, *full* dependency encompasses the *Correctness* and *Timeliness* dependencies. This relation is represented by $A \xrightarrow{full} B$ in the dependency graph.

The airway-laser surgery system design can be refined using our multi-layer dependency model. By adding a simple and reliable timer to each device we recognize that the system safety van be maintained when the supervisor- device loop becomes open. However, the supervisor is expected to send the correct commands when the loop is closed. Therefore, the system safety only depends on the correctness of supervisor software. The simple timer implemented on dedicated hardware manages to maintain safety even when the supervisor fails or the communication network disconnect. As a result of this new system composition we have the followings:

- $S \xrightarrow{full} Timer$
- $S \xrightarrow{correct} SupervisorSoftware$
- $S \xrightarrow{use} OperatingSystem$
- $S \xrightarrow{use} Microprocessor$
- $S \xrightarrow{use} CommunicationNetwork$

In this new design, the safety-critical subsystem only includes the simple timers. Therefore, differentiating the *Correctness* and *Timeliness* dependencies can result in reducing the complexity of the safety-critical subsystem by recognizing the components that the system safety depends on both their correctness and timeliness. Using the multi-layer dependency model we can compose systems with more reliable critical subsystem while satisfying all the the safety requirements. In next subsection we explain our dependency framework.

2.3 Multi-layer Dependency Model

We define system S as a graph where nodes are components annotated with fault models and annotated dependency relations are arcs connecting the components. We adopt the classic fault classification from [25]. The component faults are categorized into *value*, *crash* and *timing* faults. *Value* faults happens when given a set of inputs the component does not behave according to its specification and generates an incorrect set of outputs. *Crash* faults refers to inability of the component to respond to a set of inputs and generate the expected results. A *timing* fault occurs when a component fails to produce the results timely. The *timing* fault includes the late and

early timing failures. A *timing* fault can also represent a *crash* fault since it can be considered a *timing* fault with a delay that equals to infinity. In addition, in real-time systems, *timing* and *crash* faults have the save adverse effect on the system. Therefore, we only consider *timing* and *value* faults in our model. We use this fault model along with our refined dependency model to construct the multi-layer safety dependency graph of the system.

Given the fault model of each component in the system and the dependency relations between the components we can construct system's dependency graph. Suppose F_B denotes the set of faults that can occur in component B and $F_A^{(B)}$ denotes the set of faults in component A introduced by the faults in component B. The following theorem explains the fault propagation rules in our refined dependency graph.

Theorem1: Multi-layer Dependency Propagation Rules

1. If $A \xrightarrow{Full} B$ and $F_B = \{value, timing\}$ then $F_A^{(B)} \subseteq \{value, timing\}$
2. If $A \xrightarrow{Full} B$ and $F_B = \{value\}$ then $F_A^{(B)} \subseteq \{value, timing\}$
3. If $A \xrightarrow{Full} B$ and $F_B = \{timing\}$ then $F_A^{(B)} \subseteq \{value, timing\}$
4. If $A \xrightarrow{correct} B$ and $F_B = \{value, timing\}$ then $F_A^{(B)} \subseteq \{value, timing\}$
5. If $A \xrightarrow{correct} B$ and $F_B = \{value\}$ then $F_A^{(B)} \subseteq \{value, timing\}$
6. If $A \xrightarrow{correct} B$ and $F_B = \{timing\}$ then $F_A^{(B)} = \{\}$
7. If $A \xrightarrow{Use} B$ then $F_A^{(B)} = \{\}$

Proof: Rules 1-3 can be easily obtained form Definition 3. If component A fully depends on component B, a *timing* or *value* fault in B would trigger a fault in A. This fault can be a *timing* or *value* fault or both. The actual mapping from the fault in B to the triggered fault(s)in A depends on the way the faults are masked in A. Rules 4-6 can be derived directly from Definition 1. If component A depends on the correctness of component B, only a *value* fault in B can trigger a fault in A. The type of fault(s) triggered depends on how the faults are masked in A. However, the set of faults in A triggered by a fault in B is a subset of our fault superset, $\{value, timing\}$, in rules 1-6. Rule 7 refers to the fact that if If component A only uses the service of component B and does not depend on it, then no fault in B can trigger a fault in A.

Given the dependency graph of system S annotated with components fault model, we can determine if the system composition is safe by applying the above rules. Moreover, the optimized system composition in terms of reliability of safety-critical subsystem can be found using our refined dependency graph. Now, let us go back to our airway-laser surgery example. Fig.2.6 shows the safety dependency graph of the system.

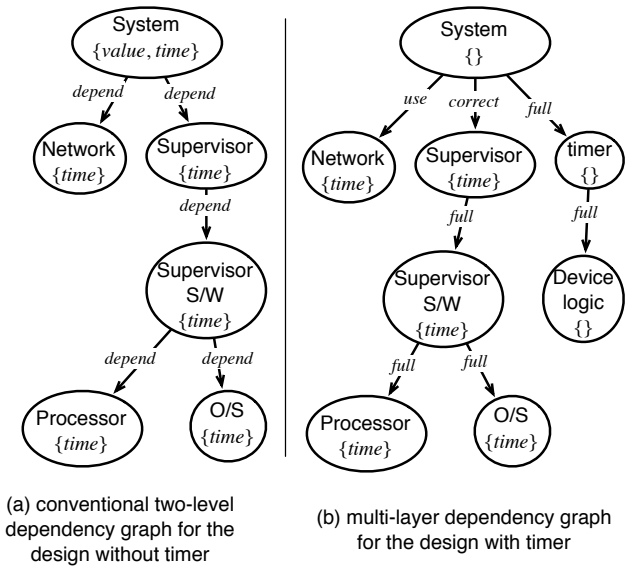


Figure 2.6. Dependency Graphs for the Airway-laser Surgery Supervision Systems

Fig.2.6.a shows dependency graph of the system with conventional design. We can see that the system safety depends on the communication network and the supervisor. We assume that the correctness of supervisor software has been verified formally. Therefore, no *value* fault can happen in this component. The microprocessor and supervisor can crash any time. As mentioned in the previous subsection, in our fault model we only consider *timing* fault which is a stronger form of *crash* fault. Therefore, the fault set of these components include the *timing* fault. Network can also disconnect and generate a *timing* fault. According to rule 3, the *timing* fault in operating system or microprocessor can trigger a *timing* or *value* fault in the supervisor software. However, in this example if the microprocessor or operating system crashes, the supervisor software will crash as well. Therefore, the *timing* fault in any of these components can only translate into a *timing* fault in the supervisor software. Ultimately, the *timing* fault in communication network or supervisor results in the failure of the system. This fact is illustrated by the non-empty set of faults in the top node which represents the system. In conclusion, this system composition is not reliable and results in violation of safety requirements.

On the contrary, the second design of airway-laser surgery system which is based on our refined dependency model results in a correct composition. As we can see from Fig.2.6.b, none of the faults occurring in the network or supervisor can translate into a system-wide failure. The system only has a full dependency on the timer implemented on dedicated hardware. According to rules 6-7, the *timing* fault in the communication network or supervisor does not translate as a system-wide fault.

In the first design, the critical part of system includes the communication network, microprocessor, operating system and the supervisor software. On the other hand, the critical subsystem in the second design only includes the

simple timer. The system safety fully depends on this component. The timer, which has a extremely simple logic, should be implemented on dedicated hardware to avoid the complex and unverifiable layers such as operating system and microprocessor. It is easy to see that the complexity and therefore the failure rate of safety-critical subsystem in the second composition is much smaller than the first composition. We adopt the exponential reliability model [26]. If λ_A and R_A are the failure rate and reliability of component A and C_A denotes its complexity, then we have the followings:

$$\lambda_A \propto C_A$$

$$R(A) \propto e^{-C_A}$$

Therefore, composing a system according to our refined multi-layer dependency model improves the reliability of safety-critical subsystem while maintaining system safety. In the next section, we evaluate the benefits of our refined dependency model by applying it to another medical application and two control systems.

Chapter 3

Case Studies

In order to demonstrate the benefits of our multi-layer dependency model, we analyze three different case studies. First, we look at the classic inverted pendulum control system and propose a new composition and compare the complexity results with those from the original Simplex design. To measure the complexity reduction in safety subsystem, we use hardware component equivalent gate count as our metric which is proportional to the complexity C in the exponential reliability model. Therefore, reducing the total gate count in the hardware component results in improvement of safety subsystem reliability. Next, we examine a rollover prevention system for autonomous off-road vehicles. After, we study the pacemaker design and evaluate the complexity reduction benefits that we can obtain by reconfiguring it using our multi-layer dependency paradigm. Finally we discuss the configuration overhead of our proposed system composition.

3.1 Cardiac Pacemaker

A cardiac pacemaker is a medical device used for patients with bradycardia condition or slow heart [27]. The device uses electric impulses to regulate the beating of the heart. However, modern pacemakers offer additional functionalities to increase the patients's comfort level. These functionalities take into account the patient's level of activity and are implemented using complex controllers. According to the reliability theory, the probability of failure of a system increases as its complexity grows. Therefore, to ensure the patient safety, Bak and et al. propose the System-level Simplex architecture in which they divide the system into a safe controller and complex controller [28]. Another safety critical component of the system is the decision module which is responsible for enforcing the safety properties. Therefore if the output from complex controller is faulty it will be rejected by the decision module. Safety controller and decision module are implemented on dedicated hardware and their correctness are verified formally. The complex controller is implemented in software. In the proposed architecture there is only two distinct levels of dependency (Fig.3.1-a).

In most of the pacemakers motion sensors are used for a better indicator of patient activity level. Complex controller use this data in addition to the data sensed from the heart to determine the output. On the other hand safety controller is a simple watchdog timer-based controller. If the complex controller is alive and its output

passes the decision module safety checks, the output rate from the complex controller will be used to to regulate the heart beat. Next, decision module will reset simple controller timer to prevent it from timing out. However, if the complex controller output is faulty the decision module does not allow it to be used. In that cased the simple controller timer will be time out and the patient heart be regulated using safety controller. In this time periods, the patient will experience some degree of discomfort.

We argue that system safety only depends on the correctness of decision logic while it fully depends on the simple controller. By adding a very simple component, which we call safety interface, to the critical subsystem we can reduce the complexity of this subsystem while maintaining safety. In the new composition, the simple controller and the safety interface make the critical part of the system. These two components are implemented in dedicated hardware. The decision logic and complex controller are implemented in software.

Fig.3.1-b shows the new pacemaker system composition. Decision logic checks the safety of the complex controller output before sending it to the critical subsystem. In the critical subsystem, the safety interface chooses the complex controller output as the output rate if a valid complex arete is received on time. Otherwise, the simple controller output is used as the output rate. The safety interface includes an error detection logic to ensure the integrity of the control outputs received across transmission channel from the decision software. We assume that decision logic augments the complex rate with some type of error detection code such as CRC-16. Therefore, if the complex rate is altered during transmission it can be detected by the interface.

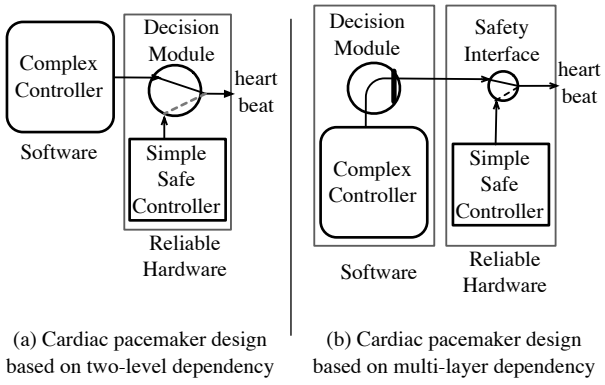


Figure 3.1. Cardiac Pacemaker Alternative System Compositions

The graph in Fig.3.2 illustrates the conventional dependency model and Fig.3.3 represents the multi-layer dependency relation between the components of pacemaker system. Each component in the graph is annotated with its fault model where *value*, and *time* represents *value*, and *timing* faults respectively. The two different system compositions are correct. However, the critical subsystem in the second composition has a lower complexity and is more reliable.

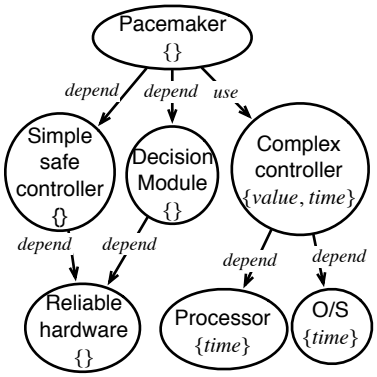


Figure 3.2. Pacemaker Conventional Dependency Graph

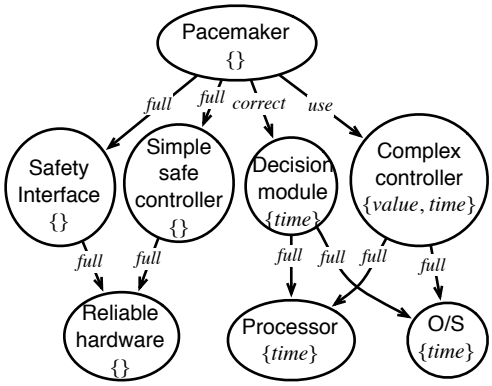


Figure 3.3. Pacemaker Multi-layer Dependency Graph

We evaluated hardware complexity of the critical subsystem in the two compositions. In the first composition in which the system is configured based on dichotomy dependency model, this subsystem includes simple controller and decision module. In the second composition which is based on multi-layer dependency model, the critical subsystem is composed of simple controller and safety interface. As mentioned earlier, the safety-critical subsystem is implemented on a dedicated hardware. The total equivalent gate count is used as the complexity metric. Both designs were synthesized for execution on a Xilinx [29] XC4VLX25 FPGA. As you see from Table3.1 migration of decision module from hardware to software results in 71% decrease in the total gate count in critical subsystem hardware.

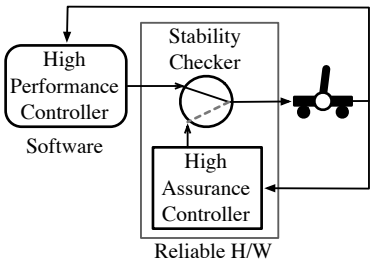
System Composition	Equivalent Gate Count
Conventional Composition	4913
Multi-layer Composition	1402

Table 3.1. Hardware Logic Utilization of Safety-critical Subsystem in Cardiac Pacemaker

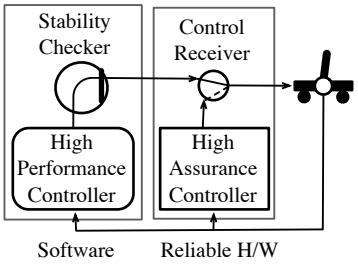
3.2 Inverted Pendulum

Let an inverted pendulum control system with Simplex architecture [13] exemplify more conventional control systems. As shown in Fig. 3.4, the configurations are quite similar to the example of cardiac pacemaker except the feedback loops. The system is composed of an inverted pendulum, a high performance controller, a high assurance controller and a stability checker. The inverted pendulum is supposed to be basically controlled by the high-performance controller. The operation of the high-performance controller is monitored by the stability checker. If the stability checker concludes that the operation of the high-performance controller may put the system unstable, the control is handed over to the high-assurance controller. The inverted pendulum's stability does not depend on the high-performance controller because the decision module continuously monitors the controller's operation. But, the stability depends on the high-assurance controller and the decision module, which are hence safety-critical components. Based on the conventional dependency model, the system-level Simplex design puts the safety-critical components in dedicated hardware for high reliability in [28].

If we adapt the multi-level dependency model, we can have a design in which the stability checker goes out of hardware component without sacrificing the reliability as shown in Fig. 3.4-b Since the reasoning dependency graphs in multi-layer dependency model resemble the ones for the pacemaker case, refer to Fig. 3.2 and 3.3. In the new design, the stability checker in software inspects the high-performance controls and block the deliveries of the potentially unsafe controls. The control reception module in hardware, which is newly added, checks the delivery of a high-performance control, and use high-assurance control if there is no delivery. Therefore, we need a simple timeout mechanism for the control reception module. In situations where the software decision module (stability



(a) Design based on conventional dependency model



(b) Design based on multi-layer dependency model

Figure 3.4. Two Configurations of Inverted Pendulum Control Systems Based on the Conventional and Multi-layer Dependency Models

checker) fails, the timer in the hardware control module times out. As a result, high-assurance controller output is used to control the inverted pendulum. In this new design, the stability has only *Correctness* dependency on the stability decision module because the high-assurance controller is employed when the decision module fails.

This new design reduces the complexity of the hardware module significantly. In Simplex architecture, the decision module is typically more complex than the high-assurance controller, so is inverted pendulum case. The stability of the system is decided by the Lyapunov function, which is composed of series of matrix multiplications, while high-assurance controller is relatively simpler. Table 3.2 compares the total equivalent gate count in hardware component when employing the conventional Simplex and the new Simplex from the multi-layer dependency model. By moving the decision module to software, less logic was necessary in hardware. Total gate count was reduced from 90418 to 12694, a reduction of about 86%. Notice that we tried to optimize the gate count of the stability checker by reusing one matrix multiplication unit for the series of matrix multiplications for fair comparisons. However, the gate count significantly reduces by moving the decision module out of the hardware component because of inherently higher complexity of dependency module than the high-assurance controller. Table3.2 shows the result of synthesizing the safety-critical hardware components for execution on a Xilinx [29] XC4VLX25 FPGA.

By moving the decision module to software, less logic was necessary in hardware. Total gate count was reduced from 90418 to 12694, a reduction of about 86%.

System Composition	Equivalent Gate Count
Conventional Composition	90418
Multi-layer Composition	12694

Table 3.2. Hardware Logic Utilization of Safety-critical Subsystem in Inverted Pendulum

3.3 Rollover Prevention System

Autonomous off-road vehicles are examples of systems which have complex software-controlled behavior, as well as physical safety properties for which verification is desirable. Although there is no operator to injure, errors in software can cause unintended actuation which can cause damage to property or the vehicle itself.

In previous work [30], we had developed a model of an off-road vehicle and applied the Simplex architectures to prevent rollover from occurring. Rollover can occur if a vehicle with a high center of gravity turns sharply uphill. More specifically, the rollover safety property was modeled as violated whenever one of the wheels of the of the vehicle left the ground, by the equation:

$$g < \frac{v^2 * \sin(\beta)}{\mathrm{WB}} * \frac{\sin\left(\tan^{-1}\left(\frac{2\mathrm{H}}{\mathrm{TR}}\right)\right)}{\cos\left(\tan^{-1}\left(\frac{2\mathrm{H}}{\mathrm{TR}}\right) + \alpha\right)}$$

g: gravitational acceleration

WB: vehicle wheel base

2H: twice the vehicle's height of center of gravity

TR: vehicle track width

v: vehicle velocity

β: vehicle steering angle

α: slope terrain angle

There is also a symmetric case for rollover where $\alpha < 0$ and $\beta < 0$. To avoid rollover, the safety controller reduces the steering angle β, and reduces the velocity v, which both matches our intuition about rollover, and is also clear from the rollover equation.

If we adapt the multi-level dependency model, we can have a design in which the stability decision module goes out of hardware module without sacrificing the reliability. We created two version of safety-critical hardware component in order to examine the amount of hardware which can be removed by moving the decision module into software. In one version, the decision module, safety controller, and external communication interface are all in hardware. In the other version, the decision module is not present in hardware. Both were synthesized for execution on a Xilinx XC4VLX25 FPGA. The comparison of these two hardware blocks in terms of FPGA logic

utilization is shown in Table 3.3.

System Composition	Occupied Slices	Gate Count
Conventional Composition	1714 (15%)	224804
Multi-layer Composition	1374 (12%)	218958

Table 3.3. Hardware Logic Utilization of Safety-critical Subsystem in Rollback Prevention System

By moving the decision module to software, less logic was necessary in hardware. The number of slices necessary was reduced from 1714 to 1374, a reduction of about 20%.

3.4 Configuration Overhead

Data transmission between two subsystems can contains errors. This can be specifically problematic when the recipient of data is the safety-critical subsystem . Therefore, some kind of error detection method should be used to prevent the invalid data being used by this subsystem. The overhead associated with error detection methods must be taken into account when calculating total complexity of the critical hardware component. In this work we studied some of the popular error detection approaches and evaluated their complexity in terms of total equivalent gate count (Table3.4).

Error Detection Method	Equivalent Gate Count
CRC16	574
CRC8	385
Hamming(39,32)	278
Repetition	149

Table 3.4. The Hardware Logic Utilization for Different Error Detection Methods

An n-bit ***cyclic redundancy check (CRC)*** applied to a data block of arbitrary length, can detect any single error burst not longer than n bits, and can detect a fraction $1 - 2^{-n}$ of all longer error bursts. We evaluated the complexity of CRC-8 and CRC-16 codes applied to a 32-bit data block. This method has a high efficiency in detecting accidental bit errors which are one of the common types of error in transmission channels. ***Hamming codes*** can detect up to two simultaneous bit errors and correct single-bit errors. For each integer $n \geq 2$ there is a code with n parity bits and $2^n - n - 1$ data bits. We measured the complexity of Hamming(39,32) code in which 7 bits of parity bits including an overall parity bit is used for a 32-bit data block. The final error detection method that we evaluated was ***Repetition codes***. In this method the data blocks are repeated across the communication channel for some pre-specified number of times. If there is a difference between the blocks received on the other side of the transmission channel, it can be determined that there is an error. We evaluated this method with two repetitions. Repetition codes are not very efficient but they can be implemented with a small amount of hardware.

Chapter 4

Distributed Medical Systems

The proposed multi-layer dependency model is specifically beneficial for safety analysis of doistributed Medical system. According to the Institute of Medicine (IOM) [31] medical accidents are the eight leading cause of death among Americans, with error-caused deaths in hospitals exceeding those from motor vehicles accidents, breast cancer or AIDs. One of such medical accidents are surgical fires. As reported by ECRI [32], which is a non-profit health agency, the basic elements of a fire are always present during the surgery. A misstep in procedure or a momentary lapse of caution can result in hazardous situations and cause considerable injury to patients. Airway-laser surgery, which is one of the surgical scenarios that can cause fire in the operating room, is the specific case study that is the focus of our work. We believe that the majority of these accidents occur due to human error. With the recent increase in the complexity of healthcare systems, stand-alone medical devices no longer provide the necessary patient care. Therefore, we need system of interoperable medical devices. The coordination of large number of medical devices by medical staff can be an error-prone process. Therefore, we need supervisory systems that can automatically manage the complex interaction of the medical devices and coordinate them in a safe manner. However, medical systems are in the safety-critical domain and the correctness of their design should be formally verified.

Such supervisory systems often include communication networks that connect all the medical devices to the central supervisor. Therefore the distributed nature of these advanced medical systems pose new challenges due to their increased complexity. High complexity means larger number of states required to be checked during the formal verification of the design. On the other hand, some of the components in the system are not verifiable. They are either too complex to be verified or the probability of their failure is too high. These components are used in medical systems or in any safety-critical system for that matter, for the advanced features they offer. However, the system safety never depends on them. The safety is often delivered by simple and formally verifiable components. They are usually implemented in hardware. This is due to the fact that hardware is more robust and less likely to fail. Moreover, if a safety-critical component is implemented in software verifying the software application by itself would not be enough. This is because the software application runs on top of an Operating System, which runs on top of a Microprocessor. Formal verification of these layers are impossible in practice due to their great

complexity. Therefore, safety of a software application can never be formally verified.

A semi-complex part of the safety component is implemented in software. This part is reliable and we depend on its correctness. The system can tolerate its failure but we require it to be a fail-safe component. In other words, its failure should not cause any harm to the other components in the system. The system expects valid output from it when it is live and working. Therefore, one the assumptions of this architecture is that failure of this component is a non-silent failure and does nor result in producing of corrupted outputs. The complex part of the system is implemented in software. This part might not be verifiable and we do not on depend on either liveness or correctness of this part.

This architecture is for design of medical systems consists of different medical devices and IT equipments in a distributed environment. NASS Framework [18] is fail-safe implementation of such architecture used for safe coordination of medical devices in a distributed environment while providing open-loop safety. This is an important factor since the supervisory control loop can break in medical systems. This can happen due to network disconnection or the crash of any common computing devices in the system.

One of the main parts of this work is introducing two types of dependencies in safety-critical embedded systems: Liveness vs. Correctness. If the system safety depends only on Correctness of a component, that component must function correctly when it is alive. This component does not have to be alive all the time and it can fail without the system safety being affected. On the other hand, if the system safety depends on Correctness *AND* Liveness of a component, the component must be alive and functioning correctly throughout the system lifetime. These components are the most robust part of the system and they maintain the system safety in spite of failure of any other parts. Therefore, they are considered safety critical part of the system. Using the conventional dependency models, it is impossible to analyze the safety of such systems.

Another important point to mention is that this supervisory system is a low-cost solution for safe coordination of distributed medical devices. This architecture allows the usage of common computing and networking equipment and COTS (Commercial Of-The-Shelf) components. This is due to the fact that in our architecture the system safety does not depend on these components. The only custom designed part of the system is the safety critical hardware component which can be implemented using very low-cost FPGA chips. The use of common equipments can also offer convenience. AS an example, we can consider Wi-Fi that can be used in clinical environments instead of complicated medical networking equipments. The safety of such cost-effective systems can be evaluated using our multi-layer dependency model. This is possible due to the differentiation of dependency types in our model. The safety of the patient can be ensured due to the fact that there is only a correctness dependency on the supervisor. The safety of this system can not be illustrated using the conventional two-layer dependency framework.

To avoid single-point failures, a flow sensor is used at the oxygen supply tube to detect the malfunction of the

computer-controlled valve. If the sensor detects that oxygen control valve is mechanically stuck close, the laser will be disabled and medical personnel will be alerted to replace the valve immediately. Flow sensor failure will be detected by the supervisory framework. Two types of failure is considered in our system. The first type is when the sensor output is stuck at a value and the second case is when the sensor stops producing values. Medical personnel will be alerted to replace the sensor immediately. It is less likely that the oxygen tank valve fails while the flow sensor being replaced. The flow sensor is a better indicator of the oxygen flow status since there is a physiological lag between when the oxygen flow is turned on and when the pulse Oximeter detects the signal. Therefore, oxygen may be present in dangerous concentration before oximeter can detect it.

Open loop safe states may not exist in various conditions. Safety means mitigation of safety hazards which are a function of the patient condition and the planned medical procedure. In the case of airway laser surgery example, if the oxygen control valve is mechanically stuck closed which will be detected by flow meter, the procedure should be stopped. In our system, in such cases, the laser is disabled and medical personnel will be alerted to replace the oxygen tank valve manually.

We analyzed the safety of NASS Framework for airway laser surgical scenario using our multi-layer dependency model and extended fault tree. In the Introduction section for the sake of simplicity and communicating the main point of this work, we referred to the safety critical component as a simple timer implemented in hardware. In the following section we discuss the main ideas behind the design and implementation of the safety critical hardware component

4.1 Safety-critical Hardware Component in NASS Framework for Airway-laser Surgery Scenario

The medical devices in NASS framework are only responsible for following the safe plans generated and sent by the the NASS supervisor to them. This is ensured by the device interface. The device interface is the component with the highest level of criticality in our supervisory system. The device interface has a very simple logic inside which is named safety logic. The safety logic has clock inside and runs a synchronization algorithm to synchronize its clock with NASS supervisor clock. Therefore, device interface does depend on any other components in the system such as the application layer running on the device host computer for clock information. Consequently, the safety logic inside the device interface can independently determine when there is failure in the system the supervisory plans do not arrive on timely manner. The failure cases include network disconnection, device host computer crash, or supervisor crash. In such cases the safety logic moves each device into a safe state. After the problem is resolved and the system is up and running, the safety logic prepares for receiving new plans by synchronizing its clock with the supervisor clock. After this step, the system is recovered to its normal operation.

For fast and convenient development of the hardware safety component of our system, we used Xilinx ML505 FPGA board. However, for the final product only the simple and small FPGA chip will be used to implement the the medical device interface. The chip logic is simple and easy to verify (refer to Appendix A for the device interface logic). There is no Operating System or application on the chip. This component is extremely robust and reliable and keeps the system in safe state when any of the common components fail. To control the medical devices through the FPGA chip we used General Purpose Input Output(GPIO) pins on the board. This method helped us to bypass any complex and unverifiable layers such as serial communication software.

There are many different kind of medical devices that are supposed to be connected into supervisory systems in hospitals. It is desirable to use the same design for different devices. In our system, device software component is highly reusable and device safety hardware component is very similar for all the devices. We used the same hardware design for oxygen controller, ventilator and surgical laser in airway-laser surgery case. Based on our implementation of NASS framework, medical device manufacturers do not have to make major modifications to their device design. The fail-safe interface of the device are taken care by our system. As mentioned before, this interface is implemented directly on hardware using a simple and small FPGA chip. The chip logic is very simple and there is no Operating System or application running on it. This component is extremely robust and reliable and keeps the system in safe sate despite the failure of any of the common components of the system.

Chapter 5

Related Work

The main idea behind our architecture is closely related to the original idea of using simplicity to control complexity [13], [14], [15], [16]. The recent increase in the use of commercial-off-the-shelf components (COTS) in distributed complex medical systems makes applying fault avoidance methods which are based on formal verification more difficult. Based on the [13], the key to improving the reliability is the existence of a simple and reliable core component that ensures the system safety despite the failure of non-core components. This approach, which is based on separation of critical requirements from desirable properties, lets us safely exploit the features and performance of complex applications. We share with [15] the idea that an architecture for design of safety-critical systems must provide application-independent utilities to tolerate hardware and software failures. In this work, the authors propose the use of their architecture for safe online upgrade and evolution of dependable systems.

The challenges of design and certifications of systems of interoperable medical devices are even harder to overcome compared to stand-alone devices. However, high confidence medical device software and systems are necessary to avoid medical errors. The authors in [22] focus on the necessity of proper infrastructures on which networks of efficient and fully functioning interoperable plug-and-play medical devices can be built. According to [23] having a systems of interconnected plug-and-play medical devices in the ICU can prevent chaotic situations where alarms from multiple devices go off. These systems can integrate the data from all the monitoring devices and proved the clinical staff with a more precise assessment of patient state. In [24] the authors also discuss the benefits of interoperable medical devices. Among the benefits mentioned at the work are the safety interlocks that would lead to more resilient systems. The seamless integration of the medical devices into a fully functional systems is only possible through plug-and-play mechanisms. The authors presents the prototype of an interoperable medical system including a PCA pump and multiple monitoring devices to discuss the existing challenges and also the benefits of such systems.

In [33], the authors use the Medical Device Coordination Framework ([34]) to develop a prototype of a PCA system. The main goal of this framework is improving he flexibility and reconfigurability of prototype multi-device medical systems and increasing the ease and speed of developing such systems. In this framework, the communication between devices happen through a flexible publish-subscribe architecture based on Java messaging

service (JMS). In the PCA system prototype a closed loop physiologic control is used to improve the safety and effectiveness of the medical care. However, throughout the work less focus has put on the safety issue which is critical in the development of medical systems. The authors do not provide a formal verification of the framework. Their proposed middleware can be too complex and very hard or even impossible to be formally verified. In case of failure of MDCF or network, they depend on the fallback behavior of the system which is as simple as the pump being turned off. Although this might be a safe solution for this scenario but as mentioned by the authors, this might not be safe for other medical scenarios.

In [35] the authors report their experience with a prototype plug-and-play medical device system that includes the interconnection of an x-ray and anesthesia machine ventilator. The proposed synchronization algorithm only works for this specific medical scenario and is not general coordination framework for other systems of interoperable medical devices. The system is built around specific devices and will not operate with different devices even with the same functionality.

Lui Sha and et. al [36] introduce a real-time logical synchrony protocol called Physically Asynchronous Logically Synchronous (PALS) to support real-rime global computation.

As the medical devices becomes more complex the cost and time spent on design, test and getting FDA approval increases for the manufacturers [22]. The challenges of design and certifications of systems of interoperable medical devices are even harder to overcome compared to stand-alone devices. However, high confidence medical device software and systems are necessary to improve the health care quality and avoid medical errors that endanger patients safety or degrade the quality of care for them. To reach this goal, we need to develop proper infrastructures on which we can build networks of efficient and fully functioning interoperable plug-and-play medical devices. Since these devices can be embedded inside patients' body, the interaction of the device with its environment should be considered in the design process. To deal with increasing complexity of the devices, the validation and certification should be incorporated in early design stages.

According to [23] having a system of interconnected plug-and-play medical devices in the ICU can prevent chaotic situations where alarms from multiple devices go off. These systems can integrate the data from all the monitoring devices and proved the clinical staff with a more precise assessment of patient state.

According to [37] the majority of the adverse events caused by PCA pumps happen during the drug administration and they are mostly attributable to user error, including program errors. In this work, the authors try to reduce this class of errors by redesigning the device interface. They believe that the complexity of user interface and programming logic of these pumps are contributing factors to these mishaps. They use human factor engineering techniques to make the PCA interface less error-prone. In this technique the representative user is involved throughout the whole design process. In addition, user-related factors such as work context, job demands,

performance bottleneck, capabilities and limitations are also considered. This method is an iterative design process that leverages user feedback to improve initial design concepts. Some of the changes made to the PCA interface by the authors include providing more feedback to user during programming and not having to rely on their memory, more straightforward programming sequence, more clear dialogue structure and message displays, and providing users with reliable shortcuts to increase the efficiency. The authors show that the new interface lead to significantly fewer total errors and better performance.

In [34] an architecture called Medical Device Coordination Framework (MDCF) is introduced for medical device integration. This framework is designed for rapid prototyping of multi-device medical systems. The main goal of this framework is improving he flexibility and reconfigurability of prototype multi-device medical systems and increasing the ease and speed of developing such systems. This framework is developed by the authors with the goal of MDCF includes a medical oriented middleware accompanied by a programming tool that hides the details of the middleware from the developers. In this framework, the communication between devices happens through a flexible publish-subscribe architecture based on Java messaging service (JMS). However, MDCF middleware is designed in a way that hides the details of underlying messaging system with providing a notion of virtual channels. These virtual channels can be established between software or physical components of medical scenario. The MDCF meta-language encapsulates the features of this abstraction. This meta-language defines MDCF programming model that can be used by the developers in the programming tool environment to design the components and the scenario composed of those components. The accompanied development tool uses a component-based programming model that abstracts away the details of underlying middleware. The three main categories of components in this model include data producers, data consumers and data transformers. Using this tool, the necessary code skeleton for each component is auto-generated. This skeleton contains the code necessary for the components to connect the runtime MDCF stack. Any logic defined by the developer is invoked by the underlying framework when new data comes. Using this framework, the developers, can focus on building efficient medical systems without having to deal with the cumbersome parts of the development process such as concurrency issues and communication codes. The authors also discuss the benefits using such framework in different clinical contexts including room-oriented device information presentation, alarm integration and forwarding, and critical care device coordination. The focus was put on the critical care device coordination since this is the most safety-critical context. They show in their experiments that the infrastructure is scalable and can support realistic deployments in clinical environments. The underlying JMS message delivery provides reliability which is specifically useful for in critical care device coordination. However, this comes with the cost of reduced performance.

In [33], the authors use the Medical Device Coordination Framework ([34]) to develop a prototype of a PCA system. In the PCA system prototype a closed loop physiologic control is used to improve the safety and effec-

tiveness of the medical care. However, throughout the work less focus has put on the safety issue which is critical in the development of medical systems. The authors do not provide a formal verification of the framework. Their proposed middleware can be too complex and very hard or even impossible to be formally verified. In case of failure of MDCF or network, they depend on the fallback behavior of the system which is as simple as the pump being turned off. Although this might be a safe solution for this scenario but as mentioned by the authors, this might not be safe for other medical scenarios.

In [35] the authors report their experience with a prototype plug-and-play medical device system. The prototype includes the interconnection of an x-ray and anesthesia machine ventilator. It is the common practice to stop the ventilator before making the exposure when an x-ray is needed during surgery. This process is necessary to take clear and high-quality x-ray images, which is only possible when the patient's lung movement is minimal. The authors propose a synchronization algorithm to find the proper time to take an x-ray image during the surgery. This interoperability framework can help prevent any sever complications as a result of not restarting the ventilator after taking the x-ray. The proposed synchronization algorithm creates a safety lock that removes the need to manually stop and restart the ventilator to take the x-ray. The authors introduce a third device called the supervisor in their system that sits between the ventilator and x-ray machine. The supervisor reads the status messages from the ventilator and decides when to trigger the x-ray machine. The synchronization algorithm running on the supervisor defines how this decision is made. The algorithm uses the respiratory cycle information to determine the time interval in which the lung movement is not significant. The authors propose two versions of the synchronization algorithm. The first version is the static version for the cases in which the respiratory frequency is not going to change between the last breath and the next one. This interval is when the sample flow rate is taken. The second version is the dynamic version in which the frequency assumption made of the first version is not valid. In this case, the real-time flow rate is sampled rapidly-enough to build a picture of the flow graph. This flow graph which represents the most recent respiratory cycle of the patient is then used to calculate the best time to trigger the x-ray. The authors build a state machine of the desired system behavior from the informal system requirements. This state machine is then verified for safety properties through model checking. In the last step, the supervisor control software code is generated from the verified state machine. The proposed synchronization algorithm only works for this specific medical scenario and is not general coordination framework for other systems of interoperable medical devices. The system is built around specific devices and will not operate with different devices even with the same functionality. In the proposed design, a translator by the LiveData Inc. is used to translate the proprietary medical device formats and makes the data available to the supervisor in XML format. SOAP standard protocol is used for communication between the supervisor and LiveData server. One of the issues with this approach is the considerable amount overhead caused by SOAP protocol. The other

issue would be the latency caused by the decoding and encoding process of XML messages. Another problem with the proposed system is that the second version of their synchronization algorithm is not practical considering the current medical devices in market. This algorithm needs a high sampling frequency form the ventilator which is not provided by the device. The correctness of the supervisor is performed through model checking EFSM of the system and generating the java code from EFSM automatically. However, they use some hand-written library functions to provide the actual value of the variables in the model. There is mention of formally verifying the correctness of these libraries in the work.

In [38]the authors propose a model-based approach for testing GUI-driven applications. GUI-driven applications are considered re-active systems that interact with a human user through a graphical user interface. As the case of study, the authors focus on a point-of-injury data entry hand-held device called AHLTA-Mobile deployed in U.S. Army. This device is used to record the patients clinical data, transmit them to a central data repository and assisting the medical personnel with diagnosis and treatment of the patients. This work focuses on the correctness a subset of this device behavior which includes multiple GUI screens displaying forms to be filled by the clinician and the last screen for entering the diagnosis. In response to user's action the data on the screen is updated or the system moves to the next screen. The authors model the system's expected behavior as a state machine in which each state represent a screen the transition between the states corresponds to changing screens. A Microsoft Research tool called NModel is used for this purpose. First, the tool library is used to create the behavioral model and the corresponding FSM for visual representation. Then, the tool's test generator is used to generate a test suite from the model. At last, the test suite is coupled to the implementation and a conformance tester is run to check for the consistency between the implementation and the model. The authors revealed an error in the AHLTA-Mobile implementation by running the conformance tester. The use of state machines to model the system might not be useful for the general dynamically modifiable GUI systems.

In [24] the authors discuss the benefits of interoperable medical devices. Among the benefits mentioned at the work are the better evaluation of the patient state through aggregated data from multiple monitoring devices and also the safety interlocks that would lead to more resilient systems. The seamless integration of the medical devices into a fully functional systems is only possible through plug-and-play mechanisms. The authors presents a prototype of interoperable medical devices to discuss the existing challenges and also the benefits of such systems. The prototype includes a PCA pump and multiple monitoring devices. The goal of this prototype system is to prevent the adverse events caused by the patient overdose. To reach this goal, a supervisor is introduced in the system. The supervisor acts as a safety interlock and stops the PCA pump when the patient's physiological data gathered from the monitoring devices fall outside a specified threshold. An alarm can also be sounded by the supervisor in these situations to get the attention of medical staff. It also decides on the drug dose based on the

recent collected data. Blood oxygen saturation (SpO2),heart rate (BPM) and simulated respiratory rate (RR) are used for the scenario. The system works on a time-triggered basis. The data is sampled form the monitoring devices on a periodic bases and the treatment dose is computed using the data and then communicate to the PCA pump. The pumps deliver the drug as long as ut receives the periodic signal form the supervisor. The authors show that depending on data from multiple monitoring devices for decision making can decrease the probability of false positives. The work also discusses the requirements for the plug-and-play medical systems to become completely feasible. One the requirements is for the medical devices to provide an external interface to remotely access the device form the supervisor. This interface should provides functionality for controlling the device and also for extracting status information. The monitoring devices interface only needs to provide status information. The second requirement is that MDPnP infrastructure must be able to parse the workflows and ensure their correct and complete execution. The underlying infrastructure also needs to provide means for computing end-to-end guarantees for delays in the system. In the presented demo, this goal is achieved by eliminating collision on the shared communication medium through temporal isolation and pre-allocated communication bandwidth. The proposed approach is not a general framework and will only operate for the represented scenario.

Chapter 6

Conclusions and Future Work

In this work, we have presented a multi-layer dependency framework in which two types of dependency relations are differentiated: Correctness vs. Liveness. Acknowledging the fact that system safety can only depend on the correctness of a component allows us to analyze the safety of many complex and critical embedded system which we can not analyze with the conventional dependency models. In addition, our multi-layer dependency framework can lead to system designs with improved reliability of safety-critical section in embedded systems. Using the refined dependency framework, we can determine if the system is safe and if the safety subsystem cost can be reduced without jeopardizing system safety.

We used a classic fault model and developed fault propagation rules for our dependency graph according to the the fault sets of the components and dependency relations between components of the system. We are planning to build a prototype toolkit based on our multi-layer dependency framework to automatically track dependencies and the propagation of faults in the system. The fault model should be tailored and augmented with more details to better suit the domain of safety-critical medical systems. We are specifically interested in medical domain with the focus on the distributed system of interoperable medical devices. The fault model should be tailored to medical domain to cover the most important and critical classes of faults in this domain.

We also plan to extend our dependency framework to decrease the pessimism inherent in such frameworks. This is due to the fact that dependency frameworks rely on worst-case analysis to ensure safety. We believe the dependency graph should help with design of safe systems while the system safety dependency is reduced to have more resilient and cost-effective systems. This approach can aid us with the design of safe and reliable systems without using expensive components by reducing the dependency of the system safety to simple and cost-effective components.

We also plan to explore more complex safety-critical embedded systems and fault-tolerant designs to illustrate the benefits of multi-layer dependency model by separating the different types of dependency in these systems. We

also like to perform availability analysis on the systems designed according to multi-layer dependency model. This is necessary to have a comprehensive understanding of the new designs. Moreover, it is critical to understand how the new dependency model can allow us to control the tradeoff between the system safety and reliability of critical subsystem on one hand and the availability of the system on the other hand.

Appendix A

VHDL Model of the Medical Device Safety Interface

It should be noted that the critical part of the model is INTR_PROC process where the plans generated by the supervisor are fed into the medical devices in a timely manner. The first section of the code is processor local bus protocol which except some minor modification is automatically generated by the Xilinx tool. We implemented our safety device interface as a device connected to Microblaze soft processor on a FPGA platform.

```
------------------------------------------------------------------------------
-- Filename:          device_interface.vhd
-- Description:       safety critical component
-- Date:              Wed Jul 21 10:31:06 2010 (by Maryam)
-- VHDL Standard:     VHDL'93
------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
library proc_common_v3_00_a;
use proc_common_v3_00_a.proc_common_pkg.all;
------------------------------------------------------------------------------
-- Entity section
------------------------------------------------------------------------------
entity user_logic is
  generic
  (C_SLV_DWIDTH                  : integer              := 32;
   C_NUM_REG                     : integer              := 31;
   C_NUM_INTR                    : integer              := 1);
  port
  (output0    : out std_logic;
   output1    : out std_logic;
   Bus2IP_Clk                    : in  std_logic;
   Bus2IP_Reset                  : in  std_logic;
   Bus2IP_Data                   : in  std_logic_vector(0 to C_SLV_DWIDTH-1);
   Bus2IP_BE                     : in  std_logic_vector(0 to C_SLV_DWIDTH/8-1);
   Bus2IP_RdCE                   : in  std_logic_vector(0 to C_NUM_REG-1);
   Bus2IP_WrCE                   : in  std_logic_vector(0 to C_NUM_REG-1);
   IP2Bus_Data                   : out std_logic_vector(0 to C_SLV_DWIDTH-1);
   IP2Bus_RdAck                  : out std_logic;
   IP2Bus_WrAck                  : out std_logic;
   IP2Bus_Error                  : out std_logic;
   IP2Bus_IntrEvent              : out std_logic_vector(0 to C_NUM_INTR-1));
end entity user_logic;
------------------------------------------------------------------------------
-- Architecture section
------------------------------------------------------------------------------
architecture IMP of user_logic is
  --USER signal declarations added here, as needed for user logic
  signal local_slv_reg0                       : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal local_slv_reg1                       : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal local_slv_reg2                       : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal local_slv_reg3                       : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal local_slv_reg4                       : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal local_slv_reg5                       : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal local_slv_reg6                       : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal local_slv_reg7                       : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal local_slv_reg8                       : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal local_slv_reg9                       : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal local_slv_reg10                      : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal local_slv_reg11                      : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal local_slv_reg12                      : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal local_slv_reg13                      : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal local_slv_reg14                      : std_logic_vector(0 to C_SLV_DWIDTH-1);
```

```
  signal local_slv_reg15                      : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal local_slv_reg16                      : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal local_slv_reg17                      : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal local_slv_reg18                      : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal local_slv_reg19                      : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal local_slv_reg20                      : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal safe_default_mode                 : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal cycle_interval               : integer;
  signal offset                    : integer;
  signal initialized                  : integer range 0 to 2 := 0;
  signal free_of_setting                : std_logic;
  signal enforcement                       : std_logic;
  signal prohibition                     : std_logic;
  signal enabled                   : std_logic;
  ---------------------------------------
  -- Signals for user logic slave model s/w accessible register example
  ---------------------------------------
  signal slv_reg0                       : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal slv_reg1                       : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal slv_reg2                       : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal slv_reg3                       : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal slv_reg4                       : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal slv_reg5                       : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal slv_reg6                       : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal slv_reg7                       : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal slv_reg8                       : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal slv_reg9                       : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal slv_reg10                      : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal slv_reg11                      : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal slv_reg12                      : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal slv_reg13                      : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal slv_reg14                      : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal slv_reg15                      : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal slv_reg16                      : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal slv_reg17                      : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal slv_reg18                      : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal slv_reg19                      : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal slv_reg20                      : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal slv_reg21                      : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal slv_reg22                      : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal slv_reg23                      : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal slv_reg24                      : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal slv_reg25                      : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal slv_reg26                      : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal slv_reg27             : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal slv_reg28             : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal slv_reg29             : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal slv_reg30             : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal slv_reg_write_sel              : std_logic_vector(0 to 30);
  signal slv_reg_read_sel               : std_logic_vector(0 to 30);
  signal slv_ip2bus_data                : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal slv_read_ack                   : std_logic;
  signal slv_write_ack                  : std_logic;
  signal intr_counter                   : std_logic_vector(0 to C_NUM_INTR-1);
begin
  slv_reg_write_sel <= Bus2IP_WrCE(0 to 30);
  slv_reg_read_sel <= Bus2IP_RdCE(0 to 30);

  slv_write_ack <= Bus2IP_WrCE(0) or Bus2IP_WrCE(1) or Bus2IP_WrCE(2) or Bus2IP_WrCE(3) or Bus2IP_WrCE(4) or
  Bus2IP_WrCE(5) or Bus2IP_WrCE(6) or Bus2IP_WrCE(7) or Bus2IP_WrCE(8) or Bus2IP_WrCE(9) or Bus2IP_WrCE(10) or
  Bus2IP_WrCE(11) or Bus2IP_WrCE(12) or Bus2IP_WrCE(13) or Bus2IP_WrCE(14) or Bus2IP_WrCE(15) or Bus2IP_WrCE(16)or
  Bus2IP_WrCE(17) or Bus2IP_WrCE(18) or Bus2IP_WrCE(19) or Bus2IP_WrCE(20) or Bus2IP_WrCE(21) or
  Bus2IP_WrCE(22) or Bus2IP_WrCE(23) or Bus2IP_WrCE(24) or Bus2IP_WrCE(25) or Bus2IP_WrCE(26) or
  Bus2IP_WrCE(27) or Bus2IP_WrCE(28) or Bus2IP_WrCE(29) or Bus2IP_WrCE(30);

  slv_read_ack <= Bus2IP_RdCE(0) or Bus2IP_RdCE(1) or Bus2IP_RdCE(2) or Bus2IP_RdCE(3) or Bus2IP_RdCE(4) or
  Bus2IP_RdCE(5) or Bus2IP_RdCE(6) or Bus2IP_RdCE(7) or Bus2IP_RdCE(8) or Bus2IP_RdCE(9) or Bus2IP_RdCE(10) or
  Bus2IP_RdCE(11) or Bus2IP_RdCE(12) or Bus2IP_RdCE(13) or Bus2IP_RdCE(14) or Bus2IP_RdCE(15) or Bus2IP_RdCE(16)or
  Bus2IP_RdCE(17)or Bus2IP_RdCE(18) or Bus2IP_RdCE(19) or Bus2IP_RdCE(20) or Bus2IP_RdCE(21) or Bus2IP_RdCE(22)or
  Bus2IP_RdCE(23) or Bus2IP_RdCE(24) or Bus2IP_RdCE(25) or Bus2IP_RdCE(26) or Bus2IP_RdCE(27)or
  Bus2IP_RdCE(28) or Bus2IP_RdCE(29) or Bus2IP_RdCE(30);

  -- implement slave model software accessible register(s)
  SLAVE_REG_WRITE_PROC : process( Bus2IP_Clk ) is
  begin
    if Bus2IP_Clk'event and Bus2IP_Clk = '1' then
      if Bus2IP_Reset = '1' then
        slv_reg0 <= (others => '0');
        slv_reg1 <= (others => '0');
        slv_reg2 <= (others => '0');
        slv_reg3 <= (others => '0');
```

```
      slv_reg4 <= (others => '0');
      slv_reg5 <= (others => '0');
      slv_reg6 <= (others => '0');
      slv_reg7 <= (others => '0');
      slv_reg8 <= (others => '0');
      slv_reg9 <= (others => '0');
      slv_reg10 <= (others => '0');
      slv_reg11 <= (others => '0');
      slv_reg12 <= (others => '0');
      slv_reg13 <= (others => '0');
      slv_reg14 <= (others => '0');
      slv_reg15 <= (others => '0');
      slv_reg16 <= (others => '0');
      slv_reg17 <= (others => '0');
      slv_reg18 <= (others => '0');
      slv_reg19 <= (others => '0');
      slv_reg20 <= (others => '0');
      slv_reg21 <= (others => '0');
      slv_reg22 <= (others => '0');
      slv_reg23 <= (others => '0');
      slv_reg24 <= (others => '0');
      slv_reg28 <= (others => '0');
  slv_reg29 <= (others => '0');
else
      case slv_reg_write_sel is
        when "10000000000000000000000000000000" =>
          for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
            if ( Bus2IP_BE(byte_index) = '1' ) then
              slv_reg0(byte_index*8 to byte_index*8+7) <= Bus2IP_Data(byte_index*8 to byte_index*8+7);
            end if;
          end loop;
        when "01000000000000000000000000000000" =>
          for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
            if ( Bus2IP_BE(byte_index) = '1' ) then
              slv_reg1(byte_index*8 to byte_index*8+7) <= Bus2IP_Data(byte_index*8 to byte_index*8+7);
            end if;
          end loop;
        when "00100000000000000000000000000000" =>
          for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
            if ( Bus2IP_BE(byte_index) = '1' ) then
              slv_reg2(byte_index*8 to byte_index*8+7) <= Bus2IP_Data(byte_index*8 to byte_index*8+7);
            end if;
          end loop;
        when "00010000000000000000000000000000" =>
          for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
            if ( Bus2IP_BE(byte_index) = '1' ) then
              slv_reg3(byte_index*8 to byte_index*8+7) <= Bus2IP_Data(byte_index*8 to byte_index*8+7);
            end if;
          end loop;
        when "00001000000000000000000000000000" =>
          for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
            if ( Bus2IP_BE(byte_index) = '1' ) then
              slv_reg4(byte_index*8 to byte_index*8+7) <= Bus2IP_Data(byte_index*8 to byte_index*8+7);
            end if;
          end loop;
        when "00000100000000000000000000000000" =>
          for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
            if ( Bus2IP_BE(byte_index) = '1' ) then
              slv_reg5(byte_index*8 to byte_index*8+7) <= Bus2IP_Data(byte_index*8 to byte_index*8+7);
            end if;
          end loop;
        when "00000010000000000000000000000000" =>
          for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
            if ( Bus2IP_BE(byte_index) = '1' ) then
              slv_reg6(byte_index*8 to byte_index*8+7) <= Bus2IP_Data(byte_index*8 to byte_index*8+7);
            end if;
          end loop;
        when "00000001000000000000000000000000" =>
          for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
            if ( Bus2IP_BE(byte_index) = '1' ) then
              slv_reg7(byte_index*8 to byte_index*8+7) <= Bus2IP_Data(byte_index*8 to byte_index*8+7);
            end if;
          end loop;
        when "00000000100000000000000000000000" =>
          for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
            if ( Bus2IP_BE(byte_index) = '1' ) then
              slv_reg8(byte_index*8 to byte_index*8+7) <= Bus2IP_Data(byte_index*8 to byte_index*8+7);
            end if;
          end loop;
        when "00000000010000000000000000000000" =>
          for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
            if ( Bus2IP_BE(byte_index) = '1' ) then
```

```
      slv_reg9(byte_index*8 to byte_index*8+7) <= Bus2IP_Data(byte_index*8 to byte_index*8+7);
    end if;
  end loop;
when "00000000001000000000000000000000" =>
  for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
    if ( Bus2IP_BE(byte_index) = '1' ) then
      slv_reg10(byte_index*8 to byte_index*8+7) <= Bus2IP_Data(byte_index*8 to byte_index*8+7);
    end if;
  end loop;
when "00000000000100000000000000000000" =>
  for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
    if ( Bus2IP_BE(byte_index) = '1' ) then
      slv_reg11(byte_index*8 to byte_index*8+7) <= Bus2IP_Data(byte_index*8 to byte_index*8+7);
    end if;
  end loop;
when "00000000000010000000000000000000" =>
  for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
    if ( Bus2IP_BE(byte_index) = '1' ) then
      slv_reg12(byte_index*8 to byte_index*8+7) <= Bus2IP_Data(byte_index*8 to byte_index*8+7);
    end if;
  end loop;
when "00000000000001000000000000000000" =>
  for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
    if ( Bus2IP_BE(byte_index) = '1' ) then
      slv_reg13(byte_index*8 to byte_index*8+7) <= Bus2IP_Data(byte_index*8 to byte_index*8+7);
    end if;
  end loop;
when "00000000000000100000000000000000" =>
  for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
    if ( Bus2IP_BE(byte_index) = '1' ) then
      slv_reg14(byte_index*8 to byte_index*8+7) <= Bus2IP_Data(byte_index*8 to byte_index*8+7);
    end if;
  end loop;
when "00000000000000010000000000000000" =>
  for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
    if ( Bus2IP_BE(byte_index) = '1' ) then
      slv_reg15(byte_index*8 to byte_index*8+7) <= Bus2IP_Data(byte_index*8 to byte_index*8+7);
    end if;
  end loop;
when "00000000000000001000000000000000" =>
  for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
    if ( Bus2IP_BE(byte_index) = '1' ) then
      slv_reg16(byte_index*8 to byte_index*8+7) <= Bus2IP_Data(byte_index*8 to byte_index*8+7);
    end if;
  end loop;
when "00000000000000000100000000000000" =>
  for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
    if ( Bus2IP_BE(byte_index) = '1' ) then
      slv_reg17(byte_index*8 to byte_index*8+7) <= Bus2IP_Data(byte_index*8 to byte_index*8+7);
    end if;
  end loop;
when "00000000000000000010000000000000" =>
  for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
    if ( Bus2IP_BE(byte_index) = '1' ) then
      slv_reg18(byte_index*8 to byte_index*8+7) <= Bus2IP_Data(byte_index*8 to byte_index*8+7);
    end if;
  end loop;
when "00000000000000000001000000000000" =>
  for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
    if ( Bus2IP_BE(byte_index) = '1' ) then
      slv_reg19(byte_index*8 to byte_index*8+7) <= Bus2IP_Data(byte_index*8 to byte_index*8+7);
    end if;
  end loop;
when "00000000000000000000100000000000" =>
  for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
    if ( Bus2IP_BE(byte_index) = '1' ) then
      slv_reg20(byte_index*8 to byte_index*8+7) <= Bus2IP_Data(byte_index*8 to byte_index*8+7);
    end if;
  end loop;
when "00000000000000000000010000000000" =>
  for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
    if ( Bus2IP_BE(byte_index) = '1' ) then
      slv_reg21(byte_index*8 to byte_index*8+7) <= Bus2IP_Data(byte_index*8 to byte_index*8+7);
    end if;
  end loop;
when "00000000000000000000001000000000" =>
  for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
    if ( Bus2IP_BE(byte_index) = '1' ) then
      slv_reg22(byte_index*8 to byte_index*8+7) <= Bus2IP_Data(byte_index*8 to byte_index*8+7);
    end if;
  end loop;
when "00000000000000000000000100000000" =>
  for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
    if ( Bus2IP_BE(byte_index) = '1' ) then
```

```
              slv_reg23(byte_index*8 to byte_index*8+7) <= Bus2IP_Data(byte_index*8 to byte_index*8+7);
            end if;
          end loop;
        when "00000000000000000000000001000000" =>
          for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
            if ( Bus2IP_BE(byte_index) = '1' ) then
              slv_reg24(byte_index*8 to byte_index*8+7) <= Bus2IP_Data(byte_index*8 to byte_index*8+7);
            end if;
          end loop;
       when "00000000000000000000000000000100" =>
           for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
            if ( Bus2IP_BE(byte_index) = '1' ) then
              slv_reg28(byte_index*8 to byte_index*8+7) <= Bus2IP_Data(byte_index*8 to byte_index*8+7);
            end if;
          end loop;
     when "00000000000000000000000000000010" =>
           for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
            if ( Bus2IP_BE(byte_index) = '1' ) then
              slv_reg29(byte_index*8 to byte_index*8+7) <= Bus2IP_Data(byte_index*8 to byte_index*8+7);
            end if;
          end loop;
     when others => null;
      end case;
    end if;
  end if;
end process SLAVE_REG_WRITE_PROC;

-- implement slave model software accessible register(s) read mux
SLAVE_REG_READ_PROC : process( slv_reg_read_sel, slv_reg0, slv_reg1, slv_reg2, slv_reg3, slv_reg4, slv_reg5,
slv_reg6, slv_reg7, slv_reg8, slv_reg9, slv_reg10, slv_reg11, slv_reg12, slv_reg13, slv_reg14, slv_reg15,
slv_reg16, slv_reg17, slv_reg18, slv_reg19, slv_reg20, slv_reg21, slv_reg22, slv_reg23, slv_reg24, slv_reg25,
slv_reg26, slv_reg27, slv_reg28, slv_reg29, slv_reg30 ) is
begin
  case slv_reg_read_sel is
    when "10000000000000000000000000000000" => slv_ip2bus_data <= slv_reg0;
    when "01000000000000000000000000000000" => slv_ip2bus_data <= slv_reg1;
    when "00100000000000000000000000000000" => slv_ip2bus_data <= slv_reg2;
    when "00010000000000000000000000000000" => slv_ip2bus_data <= slv_reg3;
    when "00001000000000000000000000000000" => slv_ip2bus_data <= slv_reg4;
    when "00000100000000000000000000000000" => slv_ip2bus_data <= slv_reg5;
    when "00000010000000000000000000000000" => slv_ip2bus_data <= slv_reg6;
    when "00000001000000000000000000000000" => slv_ip2bus_data <= slv_reg7;
    when "00000000100000000000000000000000" => slv_ip2bus_data <= slv_reg8;
    when "00000000010000000000000000000000" => slv_ip2bus_data <= slv_reg9;
    when "00000000001000000000000000000000" => slv_ip2bus_data <= slv_reg10;
    when "00000000000100000000000000000000" => slv_ip2bus_data <= slv_reg11;
    when "00000000000010000000000000000000" => slv_ip2bus_data <= slv_reg12;
    when "00000000000001000000000000000000" => slv_ip2bus_data <= slv_reg13;
    when "00000000000000100000000000000000" => slv_ip2bus_data <= slv_reg14;
    when "00000000000000010000000000000000" => slv_ip2bus_data <= slv_reg15;
    when "00000000000000001000000000000000" => slv_ip2bus_data <= slv_reg16;
    when "00000000000000000100000000000000" => slv_ip2bus_data <= slv_reg17;
    when "00000000000000000010000000000000" => slv_ip2bus_data <= slv_reg18;
    when "00000000000000000001000000000000" => slv_ip2bus_data <= slv_reg19;
    when "00000000000000000000100000000000" => slv_ip2bus_data <= slv_reg20;
    when "00000000000000000000010000000000" => slv_ip2bus_data <= slv_reg21;
    when "00000000000000000000001000000000" => slv_ip2bus_data <= slv_reg22;
    when "00000000000000000000000100000000" => slv_ip2bus_data <= slv_reg23;
    when "00000000000000000000000010000000" => slv_ip2bus_data <= slv_reg24;
    when "00000000000000000000000001000000" => slv_ip2bus_data <= slv_reg25;
    when "00000000000000000000000000100000" => slv_ip2bus_data <= slv_reg26;
    when "00000000000000000000000000010000" => slv_ip2bus_data <= slv_reg27;
    when "00000000000000000000000000001000" => slv_ip2bus_data <= slv_reg28;
    when "00000000000000000000000000000100" => slv_ip2bus_data <= slv_reg29;
    when "00000000000000000000000000000010" => slv_ip2bus_data <= slv_reg30;
    when "00000000000000000000000000000001" => slv_ip2bus_data <= slv_reg30;
    when others => slv_ip2bus_data <= (others => '0');
  end case;
end process SLAVE_REG_READ_PROC;
-----------------------------------------
INTR_PROC : process( Bus2IP_Clk ) is
   constant SIZE            : integer := 32;
constant ALL_ZERO       : std_logic_vector(0 to SIZE-1) := (others => '0');
constant NEW_DATA        : std_logic_vector(0 to SIZE-1) := (others => '1');
constant STALE_MODE   : std_logic_vector(0 to SIZE-1) := "00001111000011110000111100001111";
constant SYNCH_FAST_MODE : std_logic_vector(0 to SIZE-1) := "11111111111111110000000000000000";
constant SYNCH_SLOW_MODE : std_logic_vector(0 to SIZE-1) := "00000000000000001111111111111111";
constant INITIALIZE_MODE : std_logic_vector(0 to SIZE-1) := "00000000000000000000000000001111";
constant DISCONNECT_MODE : std_logic_vector(0 to SIZE-1) := "11110000000000000000000000000000";
constant DONE_MODE       : std_logic_vector(0 to SIZE-1) := "00001111000011110000111100001111";
constant WAITING_MODE   : std_logic_vector(0 to SIZE-1) := (others => '1');
variable reg_counter     : integer := 0;
variable cycle_counter   : integer := 0;
variable current_cycle   : integer := 0;
```

```
  begin
  if (rising_edge(Bus2IP_Clk)) then --if clock rising edge
if (initialized = 0) then
initialized <= 2;
slv_reg30 <= conv_std_logic_vector(-1,32);
current_cycle := 0;
slv_reg26 <= conv_std_logic_vector((current_cycle),32);
enforcement <= '0';
enabled <= '0';
end if;
if (slv_reg21 = STALE_MODE) then slv_reg27 <= WAITING_MODE;
elsif ((slv_reg21 = INITIALIZE_MODE) and (slv_reg27 = WAITING_MODE)) then
cycle_interval <= conv_integer(unsigned(slv_reg23));
cycle_counter := conv_integer(unsigned(slv_reg24)); --jitter
offset <= 0;
initialized <= 1;
current_cycle := conv_integer(unsigned(slv_reg28)); --cycle
local_slv_reg20 <= conv_std_logic_vector(-1,32);
reg_counter := 20;
slv_reg27 <= DONE_MODE;
elsif ((slv_reg21 = SYNCH_FAST_MODE) and (slv_reg27 = WAITING_MODE)) then
offset <= conv_integer(unsigned(slv_reg24));
slv_reg27 <= DONE_MODE;
elsif ((slv_reg21 = SYNCH_SLOW_MODE) and (slv_reg27 = WAITING_MODE)) then
offset <= -(conv_integer(unsigned(slv_reg24)));
slv_reg27 <= DONE_MODE;
elsif ((slv_reg21 = DISCONNECT_MODE) and (slv_reg27 = WAITING_MODE)) then
slv_reg30 <= conv_std_logic_vector(-1,32);
current_cycle := 0;
slv_reg26 <= conv_std_logic_vector((current_cycle),32);
slv_reg25 <= conv_std_logic_vector(0,32);
enforcement <= '0';
enabled <= '0';
initialized <= 2;
slv_reg27 <= DONE_MODE;
elsif ((slv_reg21 = NEW_DATA) and (slv_reg27 = WAITING_MODE) and
        (slv_reg22 = (current_cycle+1)) and (cycle_counter /= 0)) then
local_slv_reg0 <= slv_reg0;
local_slv_reg1 <= slv_reg1;
local_slv_reg2 <= slv_reg2;
local_slv_reg3 <= slv_reg3;
local_slv_reg4 <= slv_reg4;
local_slv_reg5 <= slv_reg5;
local_slv_reg6 <= slv_reg6;
local_slv_reg7 <= slv_reg7;
local_slv_reg8 <= slv_reg8;
local_slv_reg9 <= slv_reg9;
local_slv_reg10 <= slv_reg10;
local_slv_reg11 <= slv_reg11;
local_slv_reg12 <= slv_reg12;
local_slv_reg13 <= slv_reg13;
local_slv_reg14 <= slv_reg14;
local_slv_reg15 <= slv_reg15;
local_slv_reg16 <= slv_reg16;
local_slv_reg17 <= slv_reg17;
local_slv_reg18 <= slv_reg18;
local_slv_reg19 <= slv_reg19;
local_slv_reg20 <= slv_reg20;
slv_reg25 <= conv_std_logic_vector((current_cycle+1),32);
reg_counter:=0;
slv_reg27 <= DONE_MODE;
end if; --if flag
-------------------------------------
----------- CYCLE TICK --------------
-------------------------------------
if (initialized = 1) then
if (cycle_counter > 0) then --if counter
cycle_counter := cycle_counter - 1;
intr_counter <= (others => '0');
else
cycle_counter := cycle_interval + offset;
current_cycle := current_cycle + 1;
case reg_counter is
when 0 => slv_reg30 <= local_slv_reg0;
when 1 => slv_reg30 <= local_slv_reg1;
when 2 => slv_reg30 <= local_slv_reg2;
when 3 => slv_reg30 <= local_slv_reg3;
when 4 => slv_reg30 <= local_slv_reg4;
when 5 => slv_reg30 <= local_slv_reg5;
when 6 => slv_reg30 <= local_slv_reg6;
when 7 => slv_reg30 <= local_slv_reg7;
when 8 => slv_reg30 <= local_slv_reg8;
when 9 => slv_reg30 <= local_slv_reg9;
```

```
when 10 => slv_reg30 <= local_slv_reg10;
when 11 => slv_reg30 <= local_slv_reg11;
when 12 => slv_reg30 <= local_slv_reg12;
when 13 => slv_reg30 <= local_slv_reg13;
when 14 => slv_reg30 <= local_slv_reg14;
when 15 => slv_reg30 <= local_slv_reg15;
when 16 => slv_reg30 <= local_slv_reg16;
when 17 => slv_reg30 <= local_slv_reg17;
when 18 => slv_reg30 <= local_slv_reg18;
when 19 => slv_reg30 <= local_slv_reg19;
when others => slv_reg30 <= local_slv_reg20;
end case;
reg_counter := reg_counter + 1;
slv_reg26 <= conv_std_logic_vector((current_cycle),32);
intr_counter <= (others => '1');
            end if; --if counter
if (slv_reg30 = ALL_ZERO) then free_of_setting <= '1'; else free_of_setting <= '0';end if;
if ((slv_reg30(0) = '0') and (free_of_setting = '0')) then enforcement <= '1';
            else enforcement <= '0'; end if;
if (slv_reg30(0) = '1')  then prohibition <= '1';
            else prohibition <= '0'; end if;
if (prohibition = '1') then enabled <= '0'; end if;
if (prohibition = '0') then enabled <= '1'; end if;
end if; --if initialized
 end if; --if clock rising edge
 end process INTR_PROC;
  output0 <= enforcement;
  output1 <= enabled;
  IP2Bus_IntrEvent <= intr_counter;
  IP2Bus_Data  <= slv_ip2bus_data when slv_read_ack = '1' else (others => '0');
  IP2Bus_WrAck <= slv_write_ack;
  IP2Bus_RdAck <= slv_read_ack;
  IP2Bus_Error <= '0';
end IMP;
%\end{lstlisting}
```

References

[1] N. G. Leveson, "Software safety in embedded computer systems," *Commun. ACM*, vol. 34, pp. 34–46, February 1991.

[2] R. R. Lutz, "Analyzing Software Requirements Errors in Safety-critical, Embedded Systems," in *Proceedings of the IEEE International Symposiumon Requirements Engineering*, 1993, pp. 126–133.

[3] N. R. Storey, *Safety Critical Computer Systems.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1996.

[4] J. A. Stankovic and et al., "Strategic Directions in Real-Time and Embedded Systems," *ACM COMPUTING SURVEYS*, vol. 28, no. 4, pp. 751–763, 1996.

[5] K. C. Kapur and L. R. Lamberson, *Reliability in engineering design.* John Wiley & Sons, New York, 1977.

[6] P. D. T. O'Connor, D. Newton, and R. Bromley, *Practical Reliability Engineering (4th Ed.).* John Wiley & Sons, New York, 2002.

[7] W. S. Lee, D. L. Grosh, F. A. Tillman, and C. H. Lie, "Fault Tree Analysis, Methods, and Applications:A Review," *IEEE Transactions on Reliability*, vol. R-34, pp. 194–203, 1985.

[8] T. DeLong, "Fault Tree Manual," 1970, master's Thesis, Texas A&M University.

[9] B. Kaiser, C. Gramlich, and M. Frster, "State/Event Fault Trees- A Safety Analysis Model for Software-Controlled Systems," *Reliability Engineering & System Safety*, vol. 92, no. 11, pp. 1521 – 1537, 2007.

[10] F. Kon and R. H. Campbell, "Dependence Management in Component-Based Distributed Systems," *IEEE Concurrency*, vol. 8, pp. 26–36, 1999.

[11] Edgar Nett and Michael Mock and Peter Theisohn, "Managing dependencies - a key problem in fault-tolerant distributed algorithms," in *In Proceedings of the 27th International Symposium on Fault-Tolerant Computing*, 1997, pp. 2–10.

[12] H. Ding and L. Sha, "Dependency Algebra: A Tool for Designing Robust Real-Time Systems," in *Real-Time Systems Symposium, IEEE International*, 2005, pp. 210–220.

[13] L. Sha, "Using Simplicity to Control Complexity," *IEEE Softw.*, vol. 18, pp. 20–28, July 2001.

[14] L. Sha and J. Meseguer, "Software-Intensive Systems and New Computing Paradigms." Berlin, Heidelberg: Springer-Verlag, 2008, ch. Design of Complex Cyber Physical Systems with Formalized Architectural Patterns, pp. 92–100.

[15] L. Sha, R. Rajkumar, and M. Gagliardi, "Evolving Dependable Real-Time Systems," in *IEEE Aerospace Applications Conference*, 1995, pp. 335–346.

[16] T. L.Crenshaw, E. Gunter, C. L. Robinson, L. Sha, and P. R. Kumar, "The Simplex Reference Model: Limiting Fault-Propagation Due to Unreliable Components in Cyber-Physical System Architectures," in *Proceedings of the 28th IEEE International Real-Time Systems Symposium*, ser. RTSS '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 400–412.

[17] J. M. Goldman, "Medical Device Safety and Innovation," 2009. [Online]. Available: http://www.mdpnp.org/uploads/Capitol_Hill_NSF_CPS_MD_PnP_9July09.pdf

[18] C. Kim, M. Sun, S. Mohan, H. Yun, L. Sha, and T. F. Abdelzaher, "A framework for the safe interoperability of medical devices in the presence of network failures," in *Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems*, 2010, pp. 149–158.

[19] Http://mdpnp.cs.illinois.edu/nassvideo.html.

[20] "ASTM F2761-09: Essential safety requirements for equipment comprising the patient-centric integrated clinical environment (ICE) Part 1: General requirements and conceptual model," 2009. [Online]. Available: http://www.mdpnp.org/uploads/F2761_completed_committee_draft.pdf

[21] Http://www.mdpnp.org.

[22] I. Lee, G. J. Pappas, R. Cleavland, J. Hatcliff, B. H. Krogh, P. Lee, H. Rubin, and L. Sha, "High-Confidence Medical Device Software and Systems," *IEEE Computer*, vol. 39, pp. 33–38, April 2006.

[23] K. Grifantini, "Plug-and-Play Hospitals: Medical devices that exchange data could make hospitals safer," *MIT Technology Review*, July 2008.

[24] D. Arney, S. Fischmeister, J. M. Goldman, I. Lee, and R. Trausmuth, "Plug-and-Play for Medical Devices: Experiences from a Case Study," *Biomedical Instrumentation & Technology*, vol. 43, no. 4, pp. 313–317, 2009.

[25] F. Cristian, "Understanding fault-tolerant distributed systems," *Commun. ACM*, vol. 34, pp. 56–78, February 1991.

[26] R. E. Barlow and F. Proschan, *Mathematical theory of reliability*. John Wiley & Sons, Inc., 1965.

[27] B. Scientific, "Pacemaker System Specification," 2007. [Online]. Available: sqrl.mcmaster.ca/_SQRLDocuments/PACEMAKER.pdf

[28] S. Bak, D. K. Chivukula, O. Adekunle, M. Sun, M. Caccamo, and L. Sha, "The System-Level Simplex Architecture for Improved Real-Time Embedded System Safety," in *15th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2009, pp. 99–107.

[29] Http://www.xilinx.com.

[30] S. Bak, A. Greer, and S. Mitra, "Hybrid cyberphysical system verification with simplex using discrete abstractions," in *Proceedings of the 2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, ser. RTAS '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 143–152. [Online]. Available: http://dx.doi.org/10.1109/RTAS.2010.27

[31] Http://www.iom.edu.

[32] Http://www.ecri.org.

[33] A. King, D. Arney, I. Lee, O. Sokolsky, J. Hatcliff, and S. Procter, "Prototyping Closed Loop Physiologic Control with the Medical Device Coordination Framework," in *Proceedings of the 2010 ICSE Workshop on Software Engineering in Health Care*. ACM Special Interest Group on Software Engineering, 2010, pp. 1–11.

[34] A. King, S. Procter, D. Andresen, J. Hatcliff, S. Warren, W. Spees, R. Jetley, P. Jones, and S. Weininger, "An Open Test Bed for Medical Device Integration and Coordination," in *Proceedings of the 2009 International Conference on Software Engineering (ICSE)*, 2009, pp. 141–151.

[35] D. Arney, J. M. Goldman, S. F. Whitehead, and I. Lee, "Synchronizing an X-ray and Anesthesia Machine Ventilator: A Medical Device Interoperability Case Study," in *Proceedings of the 2009 International Conference on Biomedical Electronics and Devices (BioDevices)*, 2009, pp. 52–60.

[36] L. Sha, A. Al-nayeem, M. Sun, J. Meseguer, P. lveczky, M. Y. Nam, and P. Feiler, "PALS: Physically Asynchronous Logically Synchronous Systems," University of Illinois at Urbana-Champaign, Tech. Rep., 2009. [Online]. Available: http://hdl.handle.net/2142/11897

[37] L. Lin, K. J. Vicente, and D. J. Doyle, "Patient Safety, Potential Adverse Drug Events, and Medical Device Design: A Human Factors Engineering Approach," *Computers and Biomedical Research*, vol. 34, pp. 274–284, 2001.

[38] V. Chinnapongse, I. Lee, O. Sokolsky, S. Wang, and P. L. Jones, "Model-based Testing of GUI-Driven Applications," in *The seventh IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS 2009)*, 2009, pp. 203–214.